Linear Types and Calculi with Explicit Sharing

AUTHOR(S):

Minamide, Yasuhiko

CITATION:

Minamide, Yasuhiko. Linear Types and Calculi with Explicit Sharing. 数理解析研究所講究録 1993, 851: 88-101

ISSUE DATE:

1993-10

URL:

http://hdl.handle.net/2433/83702

数理解析研究所講究録
第 851 巻 1993 年 88-101

Linear Types and Calculi with Explicit Sharing

Yasuhiko Minamide
Research Institute for Mathematical Sciences,
Kyoto University

July 21, 1993

Abstract

We study the relation between Wadler's steadfast linear type system and calculi with explicit sharing. The calculi studied in this paper can be considered as an operational semantics of call-by-value functional programming languages. The soundness of the type system is proved and ensures that destructive operations in a well-typed program are performed safely.

1 Introduction

Many attempts have been made to introduce destructive operations into functional programming languages. One approach is introducing a data type like references of Standard ML on which destructive update can be performed. This approach is useful but violates referential transparency of programs. In another approach, the destructive operations are permitted only when they do not violate referential transparency. In order to restrict destructive operations some type systems have been proposed where only safe destructive operations are typable [GH90].

Inspired by Girard's linear logic [Gir87], Wadler proposed *steadfast linear types* [Wad90, Wad91] where destructive operations are permitted only for linear types. However, no formal proofs of the steadfast linear type system were given in [Wad90] and thus the relation between the type system and the semantics is not obvious.

In [FF89, FH92], state in untyped functional languages has been studied based on reduction semantics. In those studies, sharing in expressions is expressed by labels or variables and evaluation of expressions is defined by contextual rewriting systems.

In this paper, based on reduction semantics we prove soundness of the type system for call-by-value calculi and clarify the relation between the type system and the semantics of the calculi.

One of the most important properties of type systems is type soundness. It ensures that if a program has a type then the result of the program has the same type and establishes the relation between type systems and semantics. For pure functional programming languages soundness of type systems have been proved based on denotational semantics [Mil78]. However, for programming languages with imperative features the approach makes proofs of type soundness complicated.

On the other hand, reduction semantics clarify relation between semantics and type systems of functional programming languages extended by imperative features [WF91]. In this approach, the soundness of type systems is proved by *subject reduction* which says reduction preserves typing.

We apply the approach based on reduction semantics to the steadfast linear type system. Sharing is expressed by variables as [FH92, WF91] and soundness of the type system is proved. Type soundness of the system ensures that destructive operations in well-typed programs are safe.

A notion of read-only access was introduced in [Wad90] to use moredestructive operations. In this paper, more general type system that can treat more types as read-only is proposed. Although the first calculus presented in this paper expresses sharing naturally, it does not fit the type system extended by read-only access. Therefore we define the second calculus for the type system and then prove type soundness of the type system.

2 Untyped calculus with explicit sharing

In order to study sharing and destructive operations we use the following extended λ_v-calculus [Plo75] as [FH92, WF91]. For simplicity, shared values are restricted to only one data type which is like references of Standard ML but cannot be destructively updated.

$$\begin{array}{ll} (\mathit{Expressions}) & e ::= v \mid e_1 e_2 \mid (e_1, e_2) \mid \rho\langle x, v\rangle.e \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \\ (\mathit{Values}) & v ::= c \mid x \mid \lambda x.e \mid (v_1, v_2) \mid \texttt{ref} \mid \texttt{acc} \mid \texttt{acc!} \end{array}$$

In the expression $\rho\langle x, v\rangle.e$, $\rho\langle x, v\rangle$ represents a reference cell with value v and is referred by variable x. We use the following notation.

$$\theta ::= \{\langle x, v\rangle\}^*$$

$$\rho\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle. = \rho\langle x_1, v_1\rangle. \ldots \rho\langle x_n, v_n\rangle.$$

The operational semantics of the calculus is defined by using two relations, $\rightarrow$ and $\longmapsto$. The relation $e \rightarrow e'$ means that e is a redex reducing to e' and $e \longmapsto e'$ means that program e can be reduced to e'. The reduction relation $\rightarrow$ is defined as below. We should notice that β-reduction is performed only for values.

$$\begin{array}{lrcl} (\beta_V) & (\lambda x.e)v & \rightarrow & [v/x]e \\ (\mathit{ref}) & \texttt{ref } v & \rightarrow & \rho\langle x, v\rangle.x \\ (\mathit{deref}) & \rho\langle x, v\rangle.\rho\theta.R[\texttt{acc } x] & \rightarrow & \rho\langle x, v\rangle.\rho\theta.R[v] \\ (\mathit{deref}!) & \rho\langle x, v\rangle.\rho\theta.R[\texttt{acc! } x] & \rightarrow & \rho\theta.R[v] \\ (\rho_{\mathit{lift}}) & R[\rho\theta.e] & \rightarrow & \rho\theta.R[e] \text{ if } R \neq [] \\ (\mathit{if}) & \texttt{if } \mathit{true} \texttt{ then } e_2 \texttt{ else } e_3 & \rightarrow & e_2 \\ (\mathit{if}) & \texttt{if } \mathit{false} \texttt{ then } e_2 \texttt{ else } e_3 & \rightarrow & e_3 \end{array}$$

In rules (ref), (deref), and (ρ_{lift}) we use reference contexts defined as below.

$$R ::= [] \mid Re \mid vR \mid (R, e) \mid (v, R) \mid \texttt{if } R \texttt{ then } e_2 \texttt{ else } e_3$$

$\texttt{ref } v$ makes a reference cell,$\rho\langle x, v\rangle$, with value v referred by x. There are two way of extracting the value of a reference cell: a functional one and a destructive one. Since destructive access discards the reference cell, in order to perform `acc!` safely it is necessary that the variable referring the reference cell has exactly one reference. Reduction (ρ_{lift}) extends the scope of ρ-bound x over context R. Since context R does not contain other ρ, lifted $\rho.\langle x, v\rangle$ is not lifted over other ρs.

Evaluation of an expression, $\longmapsto$, is defined by using the evaluation context E below. An expression is evaluated from left to right.

$$E ::= [] \mid Ee \mid vE \mid \rho\theta.E \mid (E, e) \mid (v, E) \mid \texttt{if } E \texttt{ then } e_2 \texttt{ else } e_3$$

$$E[e] \longmapsto E[e'] \quad \text{if} \quad e \rightarrow e'$$

In this calculus, ordinary values are not shared, but it is sufficient because there are destructive operations only for references. The answers returned by evaluation are in form $a = \rho\langle x_1, v_1\rangle \ldots \rho\langle x_n, v_n\rangle.v$. For example, in the reduction sequence below one reference is made and then the value of the reference is destructively accessed. The destructive access is safe because there is only one y in the expression.

$$\begin{array}{lll} & (\lambda x.\texttt{acc! } x)(\texttt{ref } 1) & \\ \longmapsto & (\lambda x.\texttt{acc! } x)\rho\langle y, 1\rangle.y & \text{by } (\mathit{ref}) \\ \longmapsto & \rho\langle y, 1\rangle.(\lambda x.\texttt{acc! } x)y & \text{by } (\rho_{\mathit{lift}}) \\ \longmapsto & \rho\langle y, 1\rangle.(\texttt{acc! } y) & \text{by } (\beta_V) \\ \longmapsto & 1 & \text{by } (\mathit{deref}!) \end{array}$$

In this calculus, we cannot perform real destructive updates. We can only perform operations that reuse storage as below.

$$(\lambda y.\texttt{ref } v)(\texttt{acc! } x)$$

Although $\texttt{acc! } x$ discards x and a new reference cell is created, it is considered that it uses storage. In [GH90], a destructive update that violates referential transparency is not typable. Instead we have to reject destructive operations that cause error.

Structural Rules

(Contraction) $$\frac{\Gamma, x :!T, x :!T \vdash e : U}{\Gamma, x :!T \vdash e : U}$$

(Weakening) $$\frac{\Gamma \vdash e : U}{\Gamma, y :!T \vdash e : U}$$

Logical Rules

(var) $$\frac{}{x : T \vdash x : T}$$

(abs) $$\frac{\Gamma, x : T \vdash e : U \;\; x \notin \Gamma}{\Gamma \vdash \lambda x.e : T \multimap U}$$

(app) $$\frac{\Gamma_1 \vdash e_1 : T \multimap U \;\; \Gamma_2 \vdash e_2 : T}{\Gamma_1, \Gamma_2 \vdash e_1 e_2 : U}$$

(⊗) $$\frac{\Gamma_1 \vdash e_1 : U \;\; \Gamma_2 \vdash e_2 : V}{\Gamma_1, \Gamma_2 \vdash (e_1, e_2) : (U \otimes V)}$$

(if) $$\frac{\Gamma_1 \vdash e_1 : \;\; \Gamma_2 \vdash e_2 : \tau \;\; \Gamma_2 \vdash e_3 : \tau}{\Gamma_1, \Gamma_2 \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau}$$

Figure 1: Type Inference Rules

(!abs) $$\frac{!\Gamma, x : T \vdash e : U \;\; x \notin \Gamma}{!\Gamma \vdash \lambda x.e :!(T \multimap U)}$$

(!app) $$\frac{\Gamma_1 \vdash e_1 :!(T \multimap U) \;\; \Gamma_2 \vdash e_2 : T}{\Gamma_1, \Gamma_2 \vdash e_1 e_2 : U}$$

(!⊗) $$\frac{\Gamma_1 \vdash e_1 :!U \;\; \Gamma_2 \vdash e_2 :!V}{\Gamma_1, \Gamma_2 \vdash (e_1, e_2) :!(!U \otimes !V)}$$

Figure 2: Type Inference Rules for nonlinear types

3 Steadfast linear types

In this section we present a fragment of Wadler's steadfast linear types [Wad90, Wad91] and extend it for the calculus. The types of the steadfast linear type system are defined as below.

$$T ::= B \mid T \multimap T \mid T \otimes T \mid !T$$

where B is a base type like *int*. The type inference rules of the linear fragment of the type system are shown in Figure 1. Girard's linear logic have rules (!-R) and (Dereliction) below for the means to exchange linear and nonlinear types.

(!-R) $$\frac{!\Gamma \vdash A}{!\Gamma \vdash !A}$$

(Dereliction) $$\frac{\Gamma, A \vdash B}{\Gamma, !A \vdash B}$$

But in the presence of (!-R) and (Dereliction) it cannot be considered that values of linear types are used exactly once because by (Dereliction) linear types can be obtained from nonlinear types whose values may be referred more than once.

Since in Wadler's steadfast linear types there are no (!-R) and (Dereliction), type A and $!A$ are considered as completely distinct types. Instead of (-R) and (Dereliction), to introduce nonlinear types there are rules shown in Figure 2.

For example, by using the rules the following four types are inferred for K.

$$K = \lambda x.\lambda y.x :!(!T \multimap !(!U \multimap !T))$$
$$K = \lambda x.\lambda y.x : (!T \multimap !(!U \multimap !T))$$
$$K = \lambda x.\lambda y.x :!(T \multimap (!U \multimap T))$$
$$K = \lambda x.\lambda y.x : (T \multimap (!U \multimap T))$$

(ref) $$\frac{}{\vdash \texttt{ref} : T \multimap T\ ref}$$

(!ref) $$\frac{}{\vdash \texttt{ref} : !T \multimap !(!T\ ref)}$$

(ρ) $$\frac{\Gamma_1 \vdash v : T \quad \Gamma_2, x : T\ ref \vdash e : U}{\Gamma_1, \Gamma_2 \vdash \rho\langle x, v\rangle.e : U}$$

(ρ) $$\frac{\Gamma_1 \vdash v : !T \quad \Gamma_2, x : !(!T\ ref) \vdash e : U}{\Gamma_1, \Gamma_2 \vdash \rho\langle x, v\rangle.e : U}$$

(acc) $$\frac{}{\vdash \texttt{acc} : !(T\ ref) \multimap T}$$

(acc!) $$\frac{}{\vdash \texttt{acc!} : (T\ ref) \multimap T}$$

Figure 3: Type inference rules for references

Since we have ρ-expressions in the calculus, we need types and rules for them. The types are extended as below and the type inference rules are shown in Figure 3.

$$T ::= B \mid T \multimap T \mid T \otimes T \mid !T \mid T\ ref$$

As type assumption of ordinary variables, (Weakening) and (Contraction) of type assumption of ρ-bound variables are restricted to nonlinear cases. Therefore

$$\rho\langle x, 1\rangle.(x, x) : !(!int\ ref) \otimes !(!int\ ref)$$

but not

$$\rho\langle x, 1\rangle.(x, x) : (!int\ ref) \otimes (!int\ ref)$$

However, a reference of a linear type can be referred from more than one places as below.

$$\rho\langle x, 1\rangle.\texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ x : (int\ ref)$$

In rule **!ref**, the content of a nonlinear reference is restricted to nonlinear types. It is because if the reference is used more than once, its content may also be used more than once.

Primitive `acc` has type $T \multimap !(Tref)$ but does not have $T \multimap (Tref)$. It is because `acc` leaves $\rho\langle x, v\rangle$ even if there are no x in the expression. In rule **(acc!)** operation `acc!` can be applied only to a linear reference.

3.1 Soundness

In this section we clarify the relation between the calculus and the type system and prove type soundness of the system. Type soundness of the system ensures that destructive access in a well-typed expression is safe. The main theorem is the following.

Theorem 1 (Syntactic Soundness) *If* $\vdash e : T$ *then* $e \Uparrow$ *or* $e \longmapsto^* a$ *and* $\vdash a : T$.

$e \Uparrow$ means that for all e' such that $e \longmapsto^* e'$ there exists e'' and $e' \longmapsto e''$. This theorem is proved by using Subject Reduction Lemma below.

Lemma 2 (Subject Reduction) *If* $e \to e'$ *and* $\Gamma \vdash e : T$ *then* $\Gamma \vdash e' : T$.

Before considering the details, we shall consider some examples. If a redex in λ is reduced as below, the Subject Reduction Lemma does not hold.

$$\vdash (\lambda x.\texttt{ref}\ 1) : !(T \multimap (int\ ref))$$

$$
\begin{array}{ll}
(\lambda x.\mathbf{ref}\ 1) & \longmapsto \\
(\lambda x.\rho\langle y,1\rangle.y) & \longmapsto \\
\rho\langle y,1\rangle.(\lambda x.y) &
\end{array}
$$

The expression $\rho\langle y,1\rangle.(\lambda x.y)$ has type $(T \multimap (int\ ref))$ but does not have $!(T \multimap (int\ ref))$. Moreover β-reduction of an expression that is not a value also may cause error.

$$\vdash (\lambda x.(x,x))(\mathtt{acc!}\ (\mathbf{ref}\ 1)) : !int \otimes !int$$

$$
\begin{array}{ll}
(\lambda x.(x,x))(\mathtt{acc!}\ (\mathbf{ref}\ 1)) & \longmapsto \\
\rho\langle y,1\rangle.(\lambda x.(x,x))(\mathtt{acc!}\ y) & \longmapsto \\
\rho\langle y,1\rangle.(\mathtt{acc!}\ y, \mathtt{acc!}\ y) & \longmapsto \\
(1, \mathtt{acc!}\ y) & \longmapsto \\
Error &
\end{array}
$$

This problem occurs because $(\mathtt{acc!}\ y)$ has a nonlinear type that is inferred from a type assumption which contains linear types as below.

$$y : (!int\ ref) \vdash (\mathtt{acc!}\ y) : !int$$

However, for values nonlinear types are inferred only from nonlinear type assumptions.

Lemma 3 *If* $\Gamma \vdash v : !T$ *then* Γ *is nonlinear.*

Proof. By induction on the structure of v. □

In order to prove Substitution Lemma and Subject Reduction Lemma, we first have to prove some basic properties of the type inference. The following relation characterizes (Contraction) and (Weakening). Intuitively, $\Gamma \trianglerighteq \Gamma'$ means that Γ' can be obtained from Γ by (Contraction) and (Weakening).

Definition 4 $\Gamma \trianglerighteq \Gamma'$

- *For linear type* T,$x : T \in \Gamma$ *iff* $x : T \in \Gamma'$.
- *For nonlinear type* $!T$,*if* $x : !T \in \Gamma'$ *then* $x : !T \in \Gamma$.

For $\Gamma \trianglerighteq \Gamma'$, by using (Contraction) and (Weakening) we can obtain $\Gamma \vdash e : T$.

Lemma 5 *If* $\Gamma \trianglerighteq \Gamma'$ *and* $\Gamma' \vdash e : T$ *then* $\Gamma \vdash e : T$.

By the following lemma, we can assume the existence of the type inference whose last rule is neither (Contraction) nor (Weakening).

Lemma 6 *If* $\Gamma \vdash e : T$ *then there exists* Γ' *such that* $\Gamma \trianglerighteq \Gamma'$ *and* $\Gamma' \vdash e : T$ *whose last rule is not structural.*

Then Substitution Lemma is proved by induction on the structure of the expression and the proof is presented in the appendix.

Lemma 7 (Substitution) *If* $\Gamma_1, x : U \vdash e : T$ *and* $\Gamma_2 \vdash v : U$ *then* $\Gamma_1, \Gamma_2 \vdash [v/x]e : T$.

This lemma is used in the proof of Subject Reduction Lemma for β-reduction.

There remains the proof of Subject Reduction and we shall present essential part of the proof.
Proof of Subject Reduction Lemma.
By Lemma 5 and 6 ,we can assume that the last rule of type inference is not structural.

Case 1. (β_V) $(\lambda x.e)v \rightarrow [v/x]e$

$$\frac{\Gamma_1 \vdash \lambda x.e : !(U \multimap T)\ \ \Gamma_2 \vdash v : U}{\Gamma_1, \Gamma_2 \vdash (\lambda x.e)v : T}$$

From the definition and Lemma 6, there exists Γ_1' such that $\Gamma_1 \trianglerighteq \Gamma_1'$ and

$$\Gamma_1'; x : U \vdash e : T$$

By Substitution Lemma,

$$\Gamma_1', \Gamma_2 \vdash [v/x]e : T$$

Then by Lemma 5

$$\Gamma_1, \Gamma_2 \vdash [v/x]e : T$$

Case 2. $(deref)$ $\rho\langle x.v\rangle.\rho\theta.R[\mathtt{acc}\ x] \rightarrow \rho\langle x.v\rangle.\rho\theta.R[v]$

$$\dfrac{\Gamma_1 \vdash v :!U \quad \dfrac{\dfrac{}{x :!(!U\ ref) \vdash \mathtt{acc}\ x :!U}}{\vdots}\ \Gamma_2, x :!(!U\ ref) \vdash \rho\theta.R[\mathtt{acc}\ x] : T}{\Gamma_1, \Gamma_2 \vdash \rho\langle x, v\rangle.\rho\theta.R[\mathtt{acc}\ x] : T}$$

By Lemma 3,Γ_1 is nonlinear. Then we get the following inference.

$$\dfrac{\Gamma_1 \vdash v :!U \quad \dfrac{\dfrac{}{\Gamma_1 \vdash v :!U}}{\vdots}\ \Gamma_1, \Gamma_2, x :!(!U\ ref) \vdash \rho\theta.R[v] : T}{\dfrac{\Gamma_1, \Gamma_1, \Gamma_2 \vdash \rho\langle x, v\rangle.\rho\theta.R[v] : T}{\Gamma_1, \Gamma_2 \vdash \rho\langle x, v\rangle.\rho\theta.R[v] : T}}$$

It is formally proved by induction on the structure of R.

Case 3. $(deref!)$ $\rho\langle x.v\rangle.\rho\theta.R[\mathtt{acc!}\ x] \rightarrow \rho\theta.R[v]$

$$\dfrac{\Gamma_1 \vdash v : U \quad \dfrac{\dfrac{}{x : U\ ref \vdash \mathtt{acc!}\ x : U}}{\vdots}\ \Gamma_2, x : U\ ref \vdash \rho\theta.R[\mathtt{acc!}\ x] : T}{\Gamma_1, \Gamma_2 \vdash \rho\langle x, v\rangle.\rho\theta.R[\mathtt{acc!}\ x] : T}$$

Since R context is not a form of `if` e_1 `then` R' `else` e_3 nor `if` e_1 `then` e_2 `else` R', Γ_1 does not have to be duplicated. Then we get the following inference.

$$\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash v : U}}{\vdots}}{\Gamma_1, \Gamma_2 \vdash \rho\theta.R[v] : T}$$

□

4 Read-Only access

In the steadfast linear type system there is only one reference to the value of a linear type. It is rather restrictive for using destructive operations. In order to perform destructive operations safely, it is sufficient that there is only once reference when they are performed. Many read-only accesses to the value are safe and the types of read-only accesses correspond to nonlinear types because destructive operations are not performed on values of nonlinear types. In order that read-only accesses can be used in expressions, a new form of expressions was introduced in [Wad90].

$$\texttt{let!}\ (x)y = e_1\ \texttt{in}\ e_2$$

As ordinary `let`, after the expression e_1 is evaluated and y is bound to the value, e_2 is evaluated. However the variable x is read-only in the expression e_1. Therefore, within e_1 variable x is nonlinear and within e_2 it is linear. Moreover any part of the value of x must not be carried out outside e_1. The rule for `let!` is formally presented as below.

$$\frac{\Gamma_1, x :!T \vdash e_1 : U \quad \Gamma_2, x : T, y : U \vdash e_2 : V}{\Gamma_1, \Gamma_2, x : T \vdash \texttt{let!}\ (x)y = e_1\ \texttt{in}\ e_2 : V}$$

To ensure that the value of x is not carried out outside e_1, the following two conditions must hold.

- Any linear subtypes of T do not occur in U.
- The type U does not $\multimap$, otherwise a value whose type does not occur in U is used in the value of e_1.
- The type T does not contain $\multimap$, otherwise x cannot be used many times.

For example the following expression is well-typed.

$$x : T\ ref \vdash \texttt{let!}\ (x)y = \texttt{acc}\ x\ \texttt{in}\ (\texttt{acc!}\ x, y) : T \otimes T$$

Although in the rule of `let!` above, only one variale, x, is treated as read-only, we can use more general `let!` for read-only access by treating some linear assumptions as read-only in e_1. For example the following expression is also well-typed by using the generalized rule.

$$\texttt{let!}\ y = \texttt{acc}\ x\ \texttt{in}\ (\texttt{acc!}\ x, y)$$

However if the subtype of T occur in U accidentally, this rule cannot be applied. In the following example, since the types of x and `ref` (acc x) are the same type, x cannot be treated as read-only access.

$$\begin{array}{l}\texttt{let!}\ y = \texttt{ref}\ (\texttt{acc}\ x)\ \texttt{in}\ \texttt{acc!}\ x \\ x :!(!int\ ref) \vdash \texttt{ref}\ (\texttt{acc}\ x) :!(!int\ ref)\end{array}$$

Odersky [Ode92] proposed using the new modality of types, observer, for the purpose. However, we apply Baker's method [Bak90] of subscripted types to this rule: using subscripted $!_a$ instead of !. Then we can distinguish the !s introduced by this rule from other !s.

$$\begin{array}{l}\texttt{let!}\ y = \texttt{ref}\ (\texttt{acc}\ x)\ \texttt{in}\ \texttt{acc!}\ x :!_b int \\ x :!_a(!_b int\ ref) \vdash \texttt{ref}\ (\texttt{acc}\ x) :!_b(!_b int\ ref)\end{array}$$

The correctness of the typing rule is intuitively clear because there are no rules that remove !. However, it is difficult to prove the correctness of read-only access using the calculus presented before. Let us consider the following reduction.

$$\rho\langle x, v\rangle.\texttt{let!}\ y = R[\texttt{acc}\ x]\ \texttt{in}\ e \rightarrow \rho\langle x, v\rangle.\texttt{let!}\ y = R[v]\ \texttt{in}\ e$$

The type of the expression is inferred by using read-only access as below.

$$\dfrac{\Gamma_1 \vdash v : U \qquad \dfrac{\dfrac{\dfrac{}{x :!_a(!_a U\ ref) \vdash \texttt{acc}\ x :!_a U}}{\vdots}}{\Gamma_2, x :!_a(!_a U\ ref) \vdash R[\texttt{acc}\ x] : V} \quad \Gamma_3, x : U\ ref, y : V \vdash e : T}{\Gamma_2, \Gamma_3, x : U\ ref \vdash \texttt{let!}\ y = R[\texttt{acc}\ x]\ \texttt{in}\ e : T}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \rho\langle x, v\rangle.\texttt{let!}\ y = R[\texttt{acc}\ x]\ \texttt{in}\ e : T}$$

In order to infer $\Gamma_1, \Gamma_2, \Gamma_3 \vdash \rho\langle x, v\rangle.\texttt{let!}\ y = R[v]\ \texttt{in}\ e : T$, we have to duplicate Γ_1 because both v in $\rho\langle x, v\rangle$ and v in $R[v]$ need Γ_1 for typing. However, since the type U may be linear, the type assumption Γ_1 may contain linear types.

4.1 Another calculus with explicit sharing

Now we define another calculus that expresses sharing a little unnaturally, however it fits the type system with read-only access better. The expressions of the calculus are defined as below. For simplicity, we omit pairs and `if` in this section though they can be treated as before.

$$\begin{array}{ll} (\textit{Expressions}) & e ::= v \mid e_1 e_2 \mid \rho x.e \\ (\textit{Values}) & v ::= c \mid x \mid \langle x, v\rangle \mid \lambda x.e \mid \texttt{ref} \mid \texttt{acc} \mid \texttt{acc!} \\ & \rho\theta = \rho x_1 \ldots \rho x_n \end{array}$$

Though in the calculus before ρ occurs with a variable and a value, in this calculus it occurs only with a variable. Instead, we consider that all occurrences of $\langle x, v_i\rangle$ in an expression are shared. For example the expression equivalent to $\rho\langle x, v\rangle.(x, x)$ in the calculus before is the following.

$$\rho x.(\langle x, v\rangle, \langle x, v\rangle)$$

Therefore all v_i of $\langle x, v_i\rangle$ should be the same.

The reduction $\rightarrow$ is defined as below. The reduction of `acc` and `acc!` can be performed only when the variable is bound by ρ.

$$\begin{array}{lrcl} (\beta_V) & (\lambda x.e)v & \rightarrow & [v/x]e \\ (let) & \texttt{let!}\ x = v\ \texttt{in}\ e & \rightarrow & [v/x]e \\ (ref) & \texttt{ref}\ v & \rightarrow & \rho x.\langle x, v\rangle \\ (deref) & \rho x.\rho\theta.R[\texttt{acc}\ \langle x.v\rangle] & \rightarrow & \rho x.\rho\theta.R[v] \\ (deref!) & \rho x.\rho\theta.R[\texttt{acc!}\ \langle x.v\rangle] & \rightarrow & \rho\theta.R[v] \\ (\rho_{lift}) & R[\rho x.e] & \rightarrow & \rho x.R[e] \ \text{ if } \ R \neq [\,] \end{array}$$

Definitions of R,E,$\longmapsto$ are the same except for `let!` and ρ.

$$R ::= [\,] \mid Re \mid vR \mid \texttt{let!}\ x = R\ \texttt{in}\ e$$

$$E ::= [\,] \mid Ee \mid vE \mid \rho x.E \mid \texttt{let!}\ x = E\ \texttt{in}\ e$$

In this language, the value to be shared is not actually shared, instead ρ-bound variables denote sharing. In the following example, we consider that two $\langle y, 1\rangle$ are shared.

$$\begin{array}{lll} & (\lambda x.(\texttt{acc}\ x, x))(\texttt{ref}\ 1) & \\ \longmapsto & (\lambda x.(\texttt{acc}\ x, x))\rho y.\langle y, 1\rangle & \text{by } (ref) \\ \longmapsto & \rho y.(\lambda x.(\texttt{acc}\ x, x))\langle y, 1\rangle & \text{by } (\rho_{lift}) \\ \longmapsto & \rho y.(\texttt{acc}\ \langle y, 1\rangle, \langle y, 1\rangle) & \text{by } (\beta_V) \\ \longmapsto & \rho y.(1, \langle y, 1\rangle) & \text{by } (deref) \end{array}$$

4.2 Steadfast linear types with read-only access

Now we present a type system for the calculus defined in the last section. In type expressions, a family of modality $!_a, !_b, \ldots$ is used instead of unique $!$.

$$T ::= B \mid T \multimap T \mid !_a T \mid T\ ref$$

If a of $!_a$ does not matter, $!$ is used for simplicity. Except for the rules for sharing primitives, the type inference rules are the same as in Figure 1 and 2. The type constant $\mathcal{A}$ is used for typing of references and intuitively $x : \mathcal{A}$ means x is an address. Then type inference rules for sharing primitives are defined below.

$$(\rho) \qquad \frac{\Gamma, x : \mathcal{A} \vdash e : U}{\Gamma \vdash \rho x.e : U}$$

$$(!\rho) \qquad \frac{\Gamma, x :!_a \mathcal{A} \vdash e : U}{\Gamma \vdash \rho x.e : U}$$

$$\textbf{(addr)} \quad \frac{\Gamma \vdash v : T}{\Gamma, x : \mathcal{A} \vdash \langle x, v \rangle : (T\ ref)}$$

$$\textbf{(!addr)} \quad \frac{\Gamma \vdash v : !T}{\Gamma, x :!_a\mathcal{A} \vdash \langle x, v \rangle :!_a(!T\ ref)}$$

In order to introduce `let!` with read-only access formally, first-order types and nonlinearization are defined below.

Definition 8 *A type is first-order iff it does not contain* $\multimap$. *A type assumption* Γ *is first-order iff for all* x,$\Gamma(x)$ *is first-order.*

Definition 9 *Nonlinearization* R_a *by* $!_a$.

$$\begin{aligned} R_a(!T) &= T \\ R_a(int) &= !_a int \\ R_a(\mathcal{A}) &= !_a \mathcal{A} \\ R_a(U \otimes V) &= !_a(R_a(U) \otimes R_a(V)) \\ R_a(T\ \mathrm{ref}) &= !_a(R_a(T)\ \mathrm{ref}) \end{aligned}$$

For every type T,$R_a(T)$ is nonlinear and treated as read-only.

Then the type inference rule for `let!` with read-only access is defined as below.

$$\textbf{(let)} \quad \frac{R_a(\Gamma_1), \Gamma_2 \vdash e_1 : U \quad \Gamma_1, \Gamma_3, x : U \vdash e_2 : V \quad !_a \notin U}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \texttt{let!}\ x = e_1\ \texttt{in}\ e_2 : V}$$

where Γ_1 and U are first-order.

The type assumption Γ_1 are nonlinear in the expression e_1 and linear in e_2. For example, the type of the example before is inferred as the following.

$$\frac{x : R_a(!int\ ref) =!_a(!int\ ref) \vdash \texttt{acc}\ x :!int \quad x :!int\ ref, y :!int \vdash (\texttt{acc!}\ x, y) :!int \otimes !int}{x :!int\ ref \vdash \texttt{let!}\ y = \texttt{acc}\ x\ \texttt{in}\ (\texttt{acc!}\ x, y) :!int \otimes !int}$$

It is essential that Γ_1 and U are restricted to a first-order type assumption and a first-order type. If U is not first-order, the result of e_1 may contain values of types that are not subtypes of U.

Now we shall prove the lemmas that characterize values of first-order types.

Lemma 10 *Let* T *be first order and* $!_a \notin T$.
If $\Gamma \vdash v : T$ *then there exists a first-order type assumption* Γ' *such that* $\Gamma' \subseteq \Gamma$ *and* $!_a \notin \Gamma'$ *and* $\Gamma' \vdash v : T$.

Proof. By induction on the structure of T. □

Lemma 11 *Let* T *be first order.*
If $\Gamma \vdash v : T$ *then* $R_a(\Gamma) \vdash v : R_a(T)$

This lemma does not hold for function types. The following expression has a linear type $T \multimap int$ but not $!(T \multimap int)$.

$$y : \mathcal{A} \vdash (\lambda x.\texttt{acc!}\ \langle y, 1 \rangle) : T \multimap int$$

Proof. By induction on the structure of T.

Case 1. $T = (U\ ref)$

$$\textbf{(addr)} \quad \frac{\Gamma \vdash v : U}{\Gamma, x : \mathcal{A} \vdash \langle x, v \rangle : (U\ ref)}$$

By induction hypothesis,

$$R_a(\Gamma) \vdash v : R_a(U)$$

Then

$$R_a(\Gamma), x : R_a(\mathcal{A}) \vdash \langle x, v \rangle :!_a(R_a(U)\ ref) = R_a(T)$$

□

Now we shall prove that the reductions preserve types. The case for `let!` is proved by using Lemma 10 and the other cases are much simpler than before.

Lemma 12 (Subject Reduction) *If* $e \to e'$ *and* $\Gamma \vdash e : T$ *then* $\Gamma \vdash e' : T$.

Proof. We present only the key cases.

Case 1.

$$(let) \quad \texttt{let!}\ x = v\ \texttt{in}\ e_2 \quad \to \quad [x/v]e_2$$

and

$$\frac{R_a(\Gamma_1), \Gamma_2 \vdash v : U \quad \Gamma_1, \Gamma_3, x : U \vdash e_2 : T \quad !_a \notin U}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \texttt{let!}\ x = v\ \texttt{in}\ e_2 : T}$$

Let $\Gamma_1 = \Gamma_1', \Gamma_1''$ and Γ_1' is linear and Γ_1'' is nonlinear. By Lemma 10.

$$\frac{\Gamma_1'', \Gamma_2 \vdash v : U \quad \Gamma_1, \Gamma_3, x : U \vdash e_2 : T}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \texttt{let!}\ x = v\ \texttt{in}\ e_2 : T}$$

By Substitution Lemma,

$$\Gamma_1, \Gamma_2, \Gamma_3 \vdash [v/x]e_2 : T$$

Case 2.

$$(deref) \quad \rho x.\rho\theta.R[\texttt{acc}\,\langle x.v\rangle] \quad \to \quad \rho x.\rho\theta.R[v]$$

We consider the following case.

$$\dfrac{\dfrac{\dfrac{\Gamma' \vdash v : !U}{\Gamma', x :!_a \mathcal{A} \vdash \langle x, v\rangle :!_a (!U\ ref)}}{\Gamma', x :!_a \mathcal{A} \vdash \texttt{acc}\,\langle x, v\rangle : !U}}{\vdots}$$
$$\dfrac{\Gamma, x : \mathcal{A} \vdash \rho\theta.R[\texttt{acc}\,\langle x.v\rangle] : T}{\Gamma \vdash \rho x.\rho\theta.R[\texttt{acc}\,\langle x.v\rangle] : T}$$

Then we get the following inference.

$$\dfrac{\dfrac{\dfrac{\Gamma' \vdash v : !U}{\Gamma', x :!_a \mathcal{A} \vdash v : !U}}{\vdots}}{\dfrac{\Gamma, x : \mathcal{A} \vdash \rho\theta.R[\texttt{acc}\,\langle x.v\rangle] : T}{\Gamma \vdash \rho x.\rho\theta.R[\texttt{acc}\,\langle x.v\rangle] : T}}$$

□

It remains to prove Substitution Lemma. The only new case is `let!` and proved by using Lemma 11.

Lemma 13 (Substitution) *If* $\Gamma_1, x : U \vdash e : T$ *and* $\Gamma_2 \vdash v : U$ *then* $\Gamma_1, \Gamma_2 \vdash [v/x]e : T$.

Proof. By induction on the structure of e. Here we sketch the case of $e = \texttt{let!}\ y = e_1\ \texttt{in}\ e_2$ and the formal proof is presented in the appendix.

$$\frac{x : R_a(U), R_a(\Gamma_1), \Gamma_2 \vdash e_1 : V \quad x : U, \Gamma_1, \Gamma_3, y : V \vdash e_2 : T \quad !_a \notin V}{x : U, \Gamma_1, \Gamma_2, \Gamma_3 \vdash \texttt{let!}\ y = e_1\ \texttt{in}\ e_2 : T}$$

and

$$\Gamma_4 \vdash v : U$$

By Lemma 11

$$R_a(\Gamma_4) \vdash v : R_a(U)$$

Then by induction hypothesis,

$$R_a(\Gamma_4), R_a(\Gamma_1), \Gamma_2 \vdash e_1 : V$$

And also

$$\Gamma_4, \Gamma_1, \Gamma_3, y : U \vdash e_2 : T$$

Then

$$\Gamma_4, \Gamma_1, \Gamma_2, \Gamma_3 \vdash \texttt{let!}\ y = e_1\ \texttt{in}\ e_2 : T$$

□

5 Explicit and implicit garbage collection

We have considered that linear types are the types thas have one reference. Thus it is natural to consider that type T is a subtype of $!T$ and then the following rule that accepts acc with a linear argument is natural.

(acc) $$\frac{}{\vdash \texttt{acc} : (T\ ref) \multimap T}$$

However it violates Subject Reduction Lemma as below. The following expression is well-typed and reduces to $\rho\langle x, 1\rangle.\rho\langle y, x\rangle.\texttt{acc!}\ x$.

$$\vdash \rho\langle x, 1\rangle.\rho\langle y, x\rangle.\texttt{acc!}\ (\texttt{acc}\ y) : int$$

$$\rho\langle x, 1\rangle.\rho\langle y, x\rangle.\texttt{acc!}\ (\texttt{acc}\ y) \rightarrow \rho\langle x, 1\rangle.\rho\langle y, x\rangle.\texttt{acc!}\ x$$

However, the expression $\rho\langle x, 1\rangle.\rho\langle y, x\rangle.\texttt{acc!}\ x$ is not well-typed, because there are two x in the expression. In order to introduce the (acc) rule above, we must consider the reduction rule that corresponds to garbage collection as in [FH92].

$$(gc) \quad \rho\langle x, v\rangle.e \ \rightarrow_{gc} \ e \ (x \notin FV(e))$$

Then removing redundant reference cells makes the expression above well-typed.

$$\rho\langle x, 1\rangle.\rho\langle y, x\rangle.\texttt{acc!}\ x \rightarrow_{gc} \rho\langle x, 1\rangle.\texttt{acc!}\ x$$

and

$$\vdash \rho\langle x, 1\rangle.\texttt{acc!}\ x : int$$

Though we consider explicit garbage collection above, in the second calculus we can consider implicit garbage collection for linear types as below. The following expression corresponds to the example before.

$$\vdash \rho x.\rho y.\texttt{acc!}\ (\texttt{acc}\ \langle y, \langle x, 1\rangle\rangle) : int$$

$$\rho x.\rho y.\texttt{acc!}\ (\texttt{acc}\ \langle y, \langle x, 1\rangle\rangle) \rightarrow \rho x.\rho y.\texttt{acc!}\ \langle x, 1\rangle$$

However, the resulting expression is also well-typed by changing the type assumption $y : \mathcal{A}$ introduced by ρy into $y :!\mathcal{A}$.

6 Conclusion

We have presented two calculi with explicit sharing which are considered as operational semantics of call-by-value functional programming languages. Based on the first calculus, the relation of the steadfast linear type system and the semantics has been clarified and the type soundness has been proved. In order to treat read-only access we have considered the extension of steadfast linear type system. However, to establish the soundness of the type system with read-only access we have required the second calculus that is a little unnatural with respect to sharing.

References

[Bak90] Henry G. Baker. Unify and conquer (garbage, updatind, aliasing, ...) in functional languages. In *Proc. ACM Conf. Lisp and Functional Programming*, 1990.

[FF89] Matthias Felleisen and Daniel P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, Vol. 69,, 1989.

[FH92] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, Vol. 103,, 1992.

[GH90] Juan C. Guzmán and Paul Hudak. Single-threaded polymorhic lambda calculus. In *IEEE Symp. on Logic in Computer Science*, 1990.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, Vol. 50, No. 1,, 1987.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, Vol. 17,, 1978.

[Ode92] Martin Odersky. Observers for linear types. In *ESOPS '92*, LNCS 582, 1992.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, Vol. 1, No. 2,, December 1975.

[Wad90] Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, 1990.

[Wad91] Philip Wadler. Is there a use for linear logic? In *Partial Evaluation and Semantics Based Program Manipulation*, 1991.

[WF91] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical report, Department of Computer Science, Rice University, 1991. TR 91-160.

A Proof of Substitution Lemma

Before we sketch the proof of Substitution Lemma, we need to formalize the condition which we have implicitly assumed so far.

Definition 14 Γ *is regular.*

- *if* $x : T \in \Gamma$ *and* $x : U \in \Gamma$ *then* $T = U$.
- *if* $x : T \in \Gamma$ *and* T *is linear then* x *occurs once in* Γ.

Hereafter we assume that every type environments are regular. $(x : U)^+$ means sequence $x : U, \ldots, x : U$ which consists of more than one $x : U$. Then the following lemma is used to split a type assumption.

Lemma 15 *Let* $\Gamma', (x : U)^+ = \Gamma_1, \Gamma_2$. *Then one of the following three conditions hold.*

- $\Gamma_1 = \Gamma_1', (x : U)^+$ *and* $\Gamma_2 = \Gamma_2', (x : U)^+$ *and* U *is nonlinear.*
- $\Gamma_1 = \Gamma_1', (x : U)^+$ *and* $\Gamma_2 = \Gamma_2'$.
- $\Gamma_1 = \Gamma_1'$ *and* $\Gamma_2 = \Gamma_2', (x : U)^+$.

where $x : U \notin \Gamma_1'$ *and* $x : U \notin \Gamma_2'$.

For Substitution Lemma, we prove the following generalized lemma.

Lemma 16 (Substitution) *Let* Γ, Γ' *is regluar. If* $\Gamma, (x : U)^+ \vdash e : T$ *and* $\Gamma' \vdash v : U$ *then* $\Gamma, \Gamma' \vdash [v/x]e : T$.

Proof. By induction on the structure of e. By Lemma 6, there exists Γ_0 such that $\Gamma, (x : U)^+ \trianglerighteq \Gamma_0$ and $\Gamma_0 \vdash e : U$ whose last rule is not structural. If $x : T \notin \Gamma_0$ then $[v/x]e = e$ and T is nonlinear. Therefore

$$\Gamma_0, \Gamma' \vdash e = [v/x]e : T$$

Since $x : U \notin \Gamma_0$ and $\Gamma, (x : U)^+ \trianglerighteq \Gamma_0$,

$$\Gamma \trianglerighteq \Gamma_0$$

Then by Lemma 5,

$$\Gamma, \Gamma' \vdash e = [v/x]e : T$$

Ohterwise $\Gamma_0 = \Gamma_0', (x : U)^+$, hence we assume that the last rule is not structural. We present only key cases here.

Case 1. $e = y$ and $y \neq x$. Then U must be nonlinear and then Γ' is nonlinear Then

$$\Gamma, \Gamma' \vdash [v/x]y = y : T$$

Case 2. $e = x$. Then Γ is nonlinear and $T = U$.

$$\Gamma, \Gamma' \vdash [v/x]x = v : T$$

Case 3. $e = \lambda y.e_1$

Subcase 1. $x \neq y$. There are two cases and we preset the first case.

$$\frac{\Gamma, (x : U)^+, y : T \vdash e : V}{\Gamma, (x : U)^+ \vdash \lambda y.e : (T \multimap V)}$$

or

$$\frac{!\Gamma, (x :!U)^+, y : T \vdash e : V}{!\Gamma, (x :!U)^+ \vdash \lambda y.e :!(T \multimap V)}$$

By indution hypothesis,

$$\Gamma, (x : U)^+, z : T \vdash [z/y]e : V$$

Since z is fresh, $\Gamma, z : T, \Gamma'$ is regular. Then by induction hypothesis,

$$\Gamma, \Gamma', z : T \vdash [v/x][z/y]e : V$$

Then

$$\Gamma, \Gamma' \vdash \lambda z.[v/x][z/y]e : T \multimap V$$

Subcase 2. $x = y$ is not possible.

Case 4. $e = e_1 e_2$. By Lemma 6,

$$\frac{\Gamma_1 \vdash e_1 : V \multimap T \quad \Gamma_2 \vdash e_2 : V}{\Gamma_1, \Gamma_2 \vdash e_1 e_2 : T}$$

or

$$\frac{\Gamma_1 \vdash e_1 :!(V \multimap T) \quad \Gamma_2 \vdash e_2 : V}{\Gamma_1, \Gamma_2 \vdash e_1 e_2 : T}$$

We present only the cases for the above inference.

Subcase 1. $\Gamma_1 = \Gamma_1', (x : U)^+$ and $\Gamma_2 = \Gamma_2', (x : U)^+$ and U is nonlinear. By induction hypothesis,

$$\Gamma_1', \Gamma' \vdash [v/x]e_1 : V \multimap T$$

and

$$\Gamma_2', \Gamma' \vdash [v/x]e_2 : V$$

Sicne U is nonlinear, Γ' is nonlinear.

$$\Gamma_3, \Gamma_4, \Gamma' \vdash [v/x](e_1 e_2) : T$$

Subcase 2. $\Gamma_1 = \Gamma_1', (x : U)^+$ and $\Gamma_2 = \Gamma_2'$. By induction hypothesis,

$$\Gamma_1', \Gamma' \vdash [v/x]e_1 : V \multimap T$$

Then

$$\Gamma_1', \Gamma_2, \Gamma' \vdash [v/x](e_1 e_2) : T$$

Subcase 3. $\Gamma_1 = \Gamma_1'$ and $\Gamma_2 = \Gamma_2', (x : U)^+$. Similar to the above case.

□

Proof for the second calculus and the type system with read-only access.

Case 1. $e = \rho x.e$. Straighforward.

Case 2. $e = \langle x, w \rangle$. We need care for this case.

$$\frac{\Gamma \vdash w : !T}{\Gamma, y :!_a \mathcal{A} \vdash \langle y, w \rangle :!_a (!T\ ref)}$$

or

$$\frac{\Gamma \vdash w : T}{\Gamma, y : \mathcal{A} \vdash \langle y, w \rangle : (T\ ref)}$$

If $x \neq y$, then we can prove this case by induction hypothesis.
Ohterwise we have to show value v is a variable. But it is clear because etheir $\Gamma' \vdash v : \mathcal{A}$ or $\Gamma' \vdash v :!_a \mathcal{A}$ holds.

Case 3. $e = \texttt{let!}\ y = e_1\ \texttt{in}\ e_2$.

$$\frac{\Gamma_1, R_a(\Gamma_3), \vdash e_1 : V \quad \Gamma_2, \Gamma_3, y : V \vdash e_2 : T \quad !_a \notin V}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \texttt{let!}\ y = e_1\ \texttt{in}\ e_2 : T}$$

Subcase 1. $\Gamma_1 = \Gamma_1', (x : U)^+$ and $\Gamma_2 = \Gamma_2', (x : U)^+$ and U is nonlinear.

Subcase 2. $\Gamma_1 = \Gamma_1', (x : U)^+$ and $\Gamma_2 = \Gamma_2'$.

Subcase 3. $\Gamma_1 = \Gamma_1'$ and $\Gamma_2 = \Gamma_2', (x : U)^+$.

Subcase 4. U is linear $\Gamma_3 = \Gamma_3', x : U$.

From $\Gamma' \vdash v : U$, by Lemma 11

$$R_a(\Gamma') \vdash v : R_a(U)$$

Then by induction hypothesis,

$$\Gamma_1, R_a(\Gamma_3'), R_a(\Gamma') \vdash [v/x]e_1 : V$$

From $\Gamma_2, \Gamma_3', x : U, y : V \vdash e_2 : T$

$$\Gamma_2, \Gamma_3', x : U, z : V \vdash [z/y]e_2 : T$$

Then by induction hypothesis

$$\Gamma_2, \Gamma_3', z : V, \Gamma' \vdash [v/x][z/y]e_2 : T$$

Then $[v/x](\texttt{let!}\ y = e_1\ \texttt{in}\ e_2) \equiv \texttt{let!}\ z = [v/x]e_1\ \texttt{in}\ [v/x][z/y]e_2$

$$\Gamma_1, \Gamma_2, \Gamma_3', \Gamma' \vdash [v/x](\texttt{let!}\ y = e_1\ \texttt{in}\ e_2) : T$$

□